Part ONE
Class Design

Chapter ONE
Encapsulation and Immutable Classes

Exam Objectives

Implement encapsulation.
Create and use singleton classes and immutable classes.
Develop code that uses static keyword on initialize blocks, variables, methods, and classes.
Develop code that uses final keyword.

Encapsulation

As we know, Java is an object-oriented language, and all code must be inside a class.

```
class MyClass {
    String myField = "you";
    void myMethod() {
        System.out.println("Hi " + myField);
    }
}
```

Therefore, we have to create an object to use it:

```
MyClass myClass = new MyClass();
myClass.myField = "James";
myClass.myMethod();
```

One important concept in object-oriented programming languages is encapsulation, the ability to hide or protect an object's data.

Most of the time, when someone talks about encapsulation most people tend to think about private variables and public getters and setters and how overkilling this is, but actually, encapsulation is more than that, and it's helpful to create high-quality designs.

Let's consider the previous example.

First of all, it's good to hide as much as possible the internal implementation of a class. The reason, mitigate the impact of change.

For example, what if `myField` changes its type from `String` to `int` ?

```
myClass.myField = 1;
```

If we were using this attribute in fifty classes, we would have to make fifty changes in our program.

But if we hide the attribute and use a setter method instead, we could avoid the fifty changes:

```
// Hiding the attr to the outside world with the private keyword
private int myField = 0;

void setMyField(String val) { // Still accepting a String
    try {
        myField = Integer.parseInt(val);
    } catch(NumberFormatException e) {
        myField = 0;
    }
}
```

You implement this attribute hiding by using access modifiers.

Java supports four access modifiers:

- **public**
- **private**
- **protected**
- **default** (when no modifier is specified)

You can apply these modifiers to classes, attributes and methods according to the following table:

	Class/ Interface	Class		Interface	
		Attrib	Method	Attrib	Method
public	X	X	X	X	X
private		X	X		
protected		X	X		
default	X	X	X	X	X

As you can see, all the access modifiers can be applied to attributes and methods of a class, but not necessarily to their interface counterparts. Also, class and interface definitions can only have a `public` or default modifier. Why? Let's define first these access modifiers.

If something is declared as `public` , it can be accessed from any other class in our application, regardless of the package or the module in which it is defined.

If something is defined as `private` , it can only be accessed inside the class that defines it, not from other classes in the same packages and not from classes that inherit the class. `private` is the most restrictive access modifier.

If something is defined as `protected` , it can be only accessed by the class that defines it, its subclasses and classes of the same package. It doesn't matter if the subclass is in another package, which makes this modifier less restrictive than `private` .

If something doesn't have a modifier, it has default access also known as package access, because it can only be accessed by classes within the same package. If a subclass is defined in another package, it cannot see the default access attribute or method. That is the only difference with the `protected` modifier, which makes it more restrictive.

A code example may explain this better:

```
package street21;
public class House {
    protected int number;
    private String reference;

    void printNumber() {
        System.out.println("Num: " + number);
    }
    public void printInformation() {
        System.out.println("Num: " + number);
        System.out.println("Ref: " + reference);
    }
}
class BlueHouse extends House {
    public String getColor() { return "BLUE"; }
}

...

package city;
import street21.*;
class GenericHouse extends House {
    public static void main(String args[]) {
        GenericHouse h = new GenericHouse();
        h.number = 100;
        h.reference = ""; // Compile-time error
        h.printNumber(); // Compile-time error
        h.printInformation();
        BlueHouse bh = new BlueHouse(); // Compile-time error
        bh.getColor(); // Compile-time error
    }
}
```

- `h.number` compiles because this attribute is `protected` and `GenericHouse` can access it because it extends `House`.
- `h.reference` doesn't compile because this attribute is `private`.
- `h.printNumber()` doesn't compile because this method has default (package) access.
- `h.printInformation()` compiles because this method is `public`.
- `BlueHouse bh = new BlueHouse()` doesn't compile because the class has default (package) access.
- `bh.getColor()` doesn't compile because although the method is `public`, the class that contains it, it's not.

Now, can you see why some modifiers apply to certain elements, and others don't?

Would a `private` or `protected` class make sense in an object-oriented language?

How about a `private` method inside an interface where most, if not all methods, have no implementation?

Think about it.

Here's a summary of the rules:

	Members same class	Subclass same package	Subclass different package	Another class same package	Another class different package
public	X	X	X	X	X
private	X				
protected	X	X	X	X	
default	X	X		X	

What is a Singleton?

There will be times when you might want to have only one instance for a particular class. Such a class is a *singleton* class and is a design pattern.

There are some classes of Java that are written using the singleton pattern, for example:

- `java.lang.Runtime`
- `java.awt.Desktop`

In Java, broadly speaking, there are two ways to implement a singleton class:

- With a `private` constructor and a `static` factory method
- As an enum

Let's start with the private constructor way. Although it might seem simple at first, it poses many challenges, all of them related to keeping only one instance of a singleton class through all the application life cycle.

By having a `private` constructor, the singleton class makes sure that no other class creates an instance of it. A `private` method (or in this case, the constructor) can only be used inside the class that defines it.

```
class Singleton {
    private Singleton() { }
}
```

So the instance has to be created inside of the class itself.

This can be done in two ways. Using a `private static` attribute and a method to get it:

```
class Singleton {
    private static final Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton getInstance() {
        return instance;
    }
}
```

The attribute has to be `private` so no other class can use it, only through its getter.

It has to be `static` so the instance can be created when the class is loaded before anyone can use it and because `static` members belong to the class and not to a particular instance.

And it has to be `final` so a new instance cannot be created later.

A variation of this implementation is to use a `static` inner class (we will review this type of class on Chapter 3):

```
class Singleton {
    private Singleton() { }
    private static class Holder {
        private static final Singleton instance =
                new Singleton();
    }
    public static Singleton getInstance() {
        return Holder.instance;
    }
}
```

The advantage of this is that the instance won't be created until the inner class is referenced for the first time.

However, there will be times when for example, creating the object is not as simple as calling `new`, so the other way is to create the instance inside the `get` method:

```
class Singleton {
    private static Singleton instance;
    private Singleton() { }
    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
            // more code to create the instance...
        }
        return instance;
    }
}
```

The first time this method is called, the instance will be created. But with this approach, we face another problem. In a multithreading environment, if two or more threads are executing this method in parallel, there's a significant risk of ending up with multiple instances of the class.

One solution is to synchronize the method so only one thread at a time can access it.

```
class Singleton {
    private static Singleton instance;
    private Singleton() { }
    public static synchronized Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

The problem with this is that strictly speaking, is not very efficient. We only need synchronization the first time, when the instance is created, not every single time the method is called.

An improvement is to lock only the portion of the code that creates the instance. For this to work properly, we have to double check if the instance is `null`, one without locking (to check if the instance is already created), and then another one inside a `synchronized` block (to safely create the instance).

Here's how this would look:

```
class Singleton {
    private static Singleton instance;
    private Singleton() { }
    public static Singleton getInstance() {
        if(instance == null) {
            synchronized (Singleton.class) {
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

But again, this implementation is not perfect yet. This time, the problem is at the Java Virtual Machine (JVM) level. The JVM, or sometimes the compiler, can optimize the code by reordering or caching the value of variables (and not making the updates visible).

The solution is to use the `volatile` keyword, which guarantees that any read/write operation of a variable shared by many threads would be atomic and not cached.

```
class Singleton {
    private static volatile Singleton instance;
    private Singleton() { }
    public static Singleton getInstance() {
        if(instance == null) {
            synchronized (Singleton.class) {
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

As you can see, it was a lot of trouble to implement a singleton correctly when you want or have to defer the instantiation of the class until you first use it (also called lazy initialization). And we are not going to cover serialization and how to keep a singleton in a cluster.

So if you don't need it, either use the first two methods (create the instance when declaring the variable or use the holder inner class) or the easier (and recommended) way, an enumeration (enum).

We'll review enums in Chapter 5, for now, just knowing that enums are singletons is enough.

Immutable Objects

There will be other times when you might not want to modify the values or state of an object when used by multiple classes. Such an object will be an *immutable* object.

There are some immutable classes in the Java JDK, for example:

- `java.lang.String`
- Wrappers classes (like `Integer`)

Immutable objects cannot change after they are created. This means that they cannot have setter methods or `public` variables, so the only way to set its properties is through the constructor.

Immutability also means that if a method has to change the properties of the object, as the object or its values cannot change, it has to return a copy of the object with the new values (this is just how the `String` class works).

Another point to consider is inheritance. If the immutable class can be inherited, the subclass can change the methods to modify the instances of the class, so an immutable class cannot allow this.

In summary, an immutable object:

- Sets all of its properties through a constructor
- Does not define setter methods
- Declares all its attributes `private` (and sometimes `final`)
- Has a class declared `final` to prevent inheritance
- Protects access to any mutable state. For example, if it has a `List` member, either the reference cannot be accessible outside the object or a copy must be returned (the same applies if the object's content must change)

The static Keyword

Think of something `static` as something belonging to the class and not to a particular instance of that class.

If we are talking about attributes, a `static` attribute is shared across all instances of the class (because, again, it doesn't belong to an instance).

Compare the output of this code:

```
public class Example {
    private int attr = 0;

    public void printAttr() {
        System.out.println(attr);
    }

    public static void main(String args[]) {
        Example e1 = new Example();
        e1.attr = 10;
        e1.printAttr();
        Example e2 = new Example();
        e2.printAttr();
    }
}
```

Output:

```
10
0
```

To the output of the code that uses a `static` variable:

```
public class Example {
    private static int attr = 0;

    public void printAttr() {
        System.out.println(attr);
```

```
    }

    public static void main(String args[]) {
        Example e1 = new Example();
        e1.attr = 10;
        e1.printAttr();
        Example e2 = new Example();
        e2.printAttr();
    }
}
```

Output:

```
10
10
```

As you can see, the value is retained when another instance uses the `static` variable.

When talking about static methods, the method belongs to the class, which means that we don't need an instance to call it (the same applies to attributes, by the way.)

```
public class Example {
    private static int attr = 0;
    public static void printAttr() {
        System.out.println(attr);
    }
    public static void main(String args[]) {
        Example e1 = new Example();
        e1.attr = 10;
        e1.printAttr();
        // Referencing the method statically
        Example.printAttr();
    }
}
```

Output:

```
10
10
```

However, if you look closely, `printAttr` uses a `static` attribute, and that's the catch with `static` methods, they can't use instance variables, just `static` ones.

This makes perfect sense, if you can access a `static` method with just the class, there's no guarantee an instance exists, and even if it does, how do you link an attribute with its instance when you only have the class name?

Using the same logic, the keywords `super` and `this`, cannot be used either.

Static classes will be covered in Chapter 3, but there's another construct that can be marked as `static`, an initializer block.

A `static` (initializer) block looks like this:

```
public class Example {
    private static int number;

    static {
        number = 100;
    }
```

```
    ...
}
```

A block is executed when the class is initialized and in the order they are declared (if there's more than one).

Just like `static` methods, they cannot reference instance attributes, or use the keywords `super` and `this`.

In addition to that, `static` blocks cannot contain a `return` statement, and it's a compile-time error if the block cannot complete normally (for example, due to an uncaught exception).

Final Keyword

The `final` keyword can be applied to variables, methods, and classes.

When `final` is applied to variables, you cannot change the value of the variable after its initialization. These variables can be attributes (static and non-static) or parameters. Final attributes can be initialized either when declared, inside a constructor, or inside an initializer block.

```
public class Example {
    private final int number = 0;
    private final String name;
    private final int total;
    private final int id;

    {
        name = "Name";
    }

    public Example() {
        number = 1; // Compile-time error
        total = 10;
    }

    public void exampleMethod(final int id) {
        id = 5; // Compile-time error
        this.id = id; // Compile-time error
    }
}
```

When `final` is applied to a method, this cannot be overridden.

```
public class Human {
    final public void talk() { }
    public void eat() { }
    public void sleep() { }
}
...
public class Woman extends Human {
    public void talk() { } // Compile-time error
    public void eat() { }
    public void sleep() { }
}
```

In turn, when `final` is applied to a class, you cannot subclass it. This is used when you don't want someone to change the behavior of a class by subclassing it. Two examples in the JDK are `java.lang.String` and `java.lang.Integer`.

```
public final class Human {
    ...
}
...
public class Woman extends Human { // Compile-time error
    ...
}
```

In summary:

- A `final` variable can only be initialized once and cannot change its value after that.
- A `final` method cannot be overridden by subclasses.
- A `final` class cannot be subclassed.

Key Points

- Encapsulation is the ability to hide or protect an object's data. Java supports four access modifiers: `public`, `private`, `protected`, default (when nothing is specified, also called package-level).
- If something is declared as `public`, it can be accessed from any other class of our application. Any class, regardless of the package or the module it is defined.
- If something is defined as `private`, it can only be accessed inside the class that defines it. Not from other classes in the same packages and not from classes that inherit the class. `private` is the most restrictive access modifier.
- If something is defined as `protected`, it can be only accessed by the class that defines it, its subclasses and classes of the same package. It doesn't matter if the subclass is in another package, which makes this modifier, less restrictive than `private`.
- If something doesn't have a modifier, it has default access also known as package access, because it can only be accessed by classes within the same package. If a subclass is defined in another package, it cannot see the default access attribute or method. That is the only difference with the `protected` modifier, making it more restrictive.
- A singleton class guarantees that there's only one instance of the class during the lifetime of the application.
- An immutable object cannot change its values after it is created.
- An immutable object...
    - Sets all of its properties through a constructor
    - Does not define setter methods
    - Declares all its attributes `private` (and sometimes `final`)
    - Has a class declared `final` to prevent inheritance
    - Protects access to any mutable state. For example, if it has a `List` member, either the reference cannot be accessible outside the object or a copy must be returned (the same applies if the object's content must change)
- The `static` keyword can be applied to attributes, methods, blocks and classes. A `static` member belongs to the class where it is declared, not to a particular instance.
- The `final` keyword can be applied to variables (so they cannot change their value after initialized), methods (so they cannot be overridden), and classes (so they cannot be subclassed).

Self Test

1. Given:

```
public class Question_1_1 {
    private final int a; // 1
    static final int b = 1; // 2
    public Question_1_1() {
        System.out.print(a); // 3
        System.out.print(b); // 4
    }
}
```

What is the result?
A. `01`
B. Compilation fails on the line marked as `// 1`
C. Compilation fails on the line marked as `// 2`
D. Compilation fails on the line marked as `// 3`
E. Compilation fails on the line marked as `// 4`

2. Which of the following state the correct order from the more restricted modifier to the more unrestricted?
A. `private` , default, `public` , `protected`
B. `protected` , `private` , default, `public`
C. default, `protected` , `private` , `public`
D. `private` , default, `protected` , `public`

3. Given:

```
public final class Question_1_3 {
    private final int n;
    public Question_1_3() { }
    public void setN(int n) { this.n = n; }
}
```

Which of the following is true?
A. The class is immutable
B. The class is not immutable
C. Compilation fails
D. An exception occurs at runtime

4. Given:

```
public class Question_1_4 {
    private final List<Integer> list = new ArrayList<>();
    public void add() {
        list.add(0);
    }
    public static void main(String[] args) {
        Question_1_4 q = new Question_1_4();
        q.add();
    }
}
```

Which of the following is true?
A. Attribute `list` contains one element after main is executed
B. The class is immutable
C. Compilation fails
D. An exception occurs at runtime

5. Given:

```
public class Question_1_5 {
    private String s = "Hi";
```

```
    public static void main(String[] args) {
        Question_1_5 q = new Question_1_5();
        q.s = "Bye"; // 1
        System.out.println(q.s); // 2
    }
}
```

What is the result?
A. `Hi`
B. `Bye`
C. Compilation fails on the declaration marked as `// 1`
D. Compilation fails on the declaration marked as `// 2`

6. Given:

```
public class Question_1_6 {
    private static int a;
    private int b;
    static {
        a = 1;
        b = 2;
    }
    public static void main(String[] args) {
        Question_1_6 q1 = new Question_1_6();
        Question_1_6 q2 = new Question_1_6();
        q2.b = 1;
        System.out.println(q1.a + q2.b);
    }
}
```

What is the result?
A. `0`
B. `3`
C. `2`
D. Compilation fails

Open answers page

Contact me

---

02. Inheritance and Polymorphism